

Python.

Классы. Декораторы.

Лекция 6

Преподаватель: Дмитрий Косицин

Области видимости переменных

Области видимости переменных. Замыкания. Лямбда-функции. Функционалы из стандартной библиотеки.

© Dzmitryi Kasitsyn

Fall 2017 • 3

Некоторые замечания

© Dzmitryi Kasitsyn

Fall 2017 • 14

Классы в Python

Наследование. Просмотр иерархии. Специальные методы.

© Dzmitryi Kasitsyn

Fall 2017 • 18

Декораторы

© Dzmitryi Kasitsyn

Fall 2017 • 32

Ответы на вопросы

© Dzmitryi Kasitsyn

Fall 2017 • 41

Области видимости переменных

...

Области видимости переменных. Замыкания. Лямбда-функции. Функционалы из стандартной библиотеки.

Области видимости переменных

Области видимости переменных определяются функциями:

- *built-in* – встроенные общедоступные имена (доступны через модуль `builtins` или `__builtins__`, например, **sum**, **abs** и т.д.)
- *global* – переменные, определенные глобально для модуля
- *enclosing* – переменные, определенные в родительской функции
- *local* – локальные для функции переменные

Локальные переменные в функциях могут в них свободно изменяться, *enclosing*, *global* и *built-in* – только читаться.

Области видимости переменных

Пример:

```
>>> abs(2)    # built-in
>>> abs = dir  # global, overrides
>>> def f():
>>>     abs = sum    # enclosing
>>>     def g():
>>>         abs = max    # local
```

Для справки. (Нужно крайне редко). Для переопределения **abs** из функции **g** в функции **f** используется ключевое слово *nonlocal*, для переопределения глобальной переменной **abs** – ключевое слово *global*.

```
>>> global abs = max    # переопределит abs, глобальный для модуля
```

Переменные в циклах

Циклы *не имеют* своей области видимости: как только переменная была создана, она становится доступной и после цикла.

```
>>> for i in range(10):  
>>>     if i > 5:  
>>>         break  
>>> print(i)  
6
```

Замечание. В Python 3.4 и ниже так же «утекали» переменные из comprehension-выражений. Баг был исправлен в Python 3.5.

```
>>> x = [i for i in range(10)]  
>>> print(i)
```

```
NameError: name 'i' is not defined # Python 3.6
```

Замыкания

В функции доступны переменные, определенные уровнями выше – они *замыкаются*.

```
>>> def make_adder(x):  
>>>     def adder(y):  
>>>         return x + y  
>>>     return adder  
  
>>> add_five = make_adder(5)  
>>> add_five(10) # 15
```

Важно! Значение замкнутой переменной получается *каждый раз* при вычислении выражения.

Пример замыкания

```
>>> x = 2
```

```
>>> def make_adder():  
>>>     def adder(y):  
>>>         return x + y  
>>>     return adder
```

```
>>> add_x = make_adder()  
>>> add_x(-2)    # 0
```

```
>>> del x    # delete 'x' - unbind variable name with object  
>>> add_x(-2)    # NameError: name 'x' is not defined
```


Statement *del*

Statement **del** имеет несколько смысловых нагрузок:

- «разрывает связь» между переменной и объектом (здесь также задействуется счетчик ссылок объекта)
- удаляет элемент, атрибут или слайс (за удаление отвечает сам объект)

```
>>> class X(object):  
>>>     a = None  
>>>  
>>> del X.a  
>>>  
>>> y, z = [1, 2, 3], {'x': 0}  
  
>>> del y[0], z['x']
```

Вопрос: как с помощью **del** очистить список, не удалив при этом объект?

Lambda-функции

Lambda-функции в Python допускают в себе одно лишь выражение:
lambda arguments: expression

Эквивалентно:

```
def <lambda>(arguments):  
    return expression
```

Вопрос: как будет выглядеть **lambda**, которая ничего не принимает и не возвращает?

Пример lambda-функций

Функция, возвращающая сумму аргументов:

```
>>> lambda x, y: x + y
```

Пример списка lambda-функций:

```
>>> collection_of_lambdas = [lambda: i*i for i in range(6)]  
>>>  
>>> for f in collection_of_lambdas:  
>>>     print(f())
```

Вопрос: что будет выведено в результате выполнения?

Пример lambda-функций

Поскольку вычисление происходит run-time, то для всех созданных функций значение `i` будет равно 5. Переменная `i` была «захвачена» в comprehension-выражении, хоть и вне этого выражения она недоступна (в Python 3.6).

Для «захвата» значения можно создать локальную для **lambda** копию:

```
>>> lambdas = [lambda i=i: i*i for i in range(6)]
```

В модуле **operator** ([Py2](#), [Py3](#)) есть множество функционалов, которыми можно пользоваться наряду с **lambda**-функциями:

```
>>> import operator
>>> # аналог: lambda x, y: x + y
>>> operator.add # operator.__add__
```

Замечания по операторам

Помимо операторов арифметических операций и операций сравнения, есть функционалы для работы с атрибутами и элементами коллекций.

```
f = operator.attrgetter('name.first', 'name.last')  
# the call f(b) returns (b.name.first, b.name.last)
```

```
g = operator.itemgetter(2, 5, 3)  
# the call g(r) returns (r[2], r[5], r[3])
```

```
h = operator.methodcaller('name', 'foo', bar=1)  
# the call h(b) returns b.name('foo', bar=1)
```

Функционалы и `lambda`-функции наряду с обычными функциями используются как аргументы, выполняющие некоторое действие, например, в **map**, **filter**.

Некоторые замечания

...

СИНГЛТОНЫ

Помимо **None**, **True**, **False**, **NotImplemented** и **Ellipsis**, числа от **-5** до **256** могут быть также синглтонами ввиду их частого использования. Это зависит от *реализации интерпретатора*.

Самый простой способ создать синглтон – объявить его глобальным в модуле, как это было сделано с **not_set**: при каждой загрузке модуля объект создаваться заново не будет.

```
>>> not_set = object()
>>> def f(x=not_set):
>>>     is_x_set = (x is not not_set)
```

Спецификация функции действием

Для спецификации функции действием не стоит передавать в нее строку или число, которое указывает на тип действия (пример – “sort”/”random”)

Флаг обычно используют для того, чтобы уточнить делать некоторую операцию или нет: **do_transform=True**. В качестве простого решения флаг использовать можно.

Если вариантов много, решение может быть в использовании Enum (в стандартной библиотеке в Py3.4+, либо сторонняя библиотека **enum**).

```
class OperationType(enum.Enum):  
    SORT = enum.auto()  
    SHUFFLE = enum.auto()
```


Спецификация функции действием

Тогда в функции выбор действия будет выглядеть так:

```
if operation_type == OperationType.SORT:  
    do_sort_transform()  
elif operation_type == OperationType.SHUFFLE:  
    do_shuffle_transform()  
else:  
    raise Exception('incorrect operation type: %s'  
                    % operation_type)
```

Другим вариантом решения будет передача некоторого функционала *transform* непосредственно в функцию – останется только его применить.

Классы в Python

...

Наследование. Просмотр иерархии. Специальные методы.

Пример наследования

```
class Animal(object):  
    pass
```

```
class Cat(Animal):  
    pass
```

```
class Dog(Animal):  
    pass
```

```
bob = Cat()
```

Проверка типа

Проверка типа с учетом наследования производится с помощью функции **isinstance**:

```
>>> isinstance(bob, Cat)    # True
>>> isinstance(bob, Animal) # True
>>> isinstance(bob, Dog)    # False
>>> type(bob) is Animal    # False; type is Cat
```

Все объекты наследуются от **object**:

```
>>> isinstance(bob, object) # True
```

Замечание. Метод **isinstance** вторым аргументом принимает также *tuple* допустимых типов.

Замечание. Для корректной проверки, является ли объект *x* целым числом, следует использовать **isinstance(x, (int, long))**.

Иерархия наследования

Посмотреть иерархию наследования можно с помощью метода **mro()** или атрибута **__mro__**:

```
>>> Cat.mro()  
[<class '__main__.Cat'>, <class '__main__.Animal'>, <class 'object'>]
```

Для проверки того, что некоторый класс является подклассом другого класса, используется функция **issubclass**:

```
>>> issubclass(Cat, Animal)    # True
```

Наследование методов и атрибутов

Наследование классов позволяет не переписывать некоторые общие для подклассов методы, оставив их в базовом классе.

```
>>> class A(object):
>>>     X = 1
>>>     def f(self):
>>>         print("Called A.f()")
>>>
>>> class B(A):
>>>     pass
>>>
>>> b = B()
>>> b.f()
>>> print(b.X)
Called A.f()
1
```

Переопределение атрибутов

Поведение в дочернем классе, разумеется, можно переопределить.

```
>>> class A(object):
>>>     def f(self):
>>>         print("Called A.f()")
>>>
>>> class B(A):
>>>     def f(self):
>>>         print("Called B.f()")
>>>
>>> a, b = A(), B()
>>> a.f()
>>> b.f()
Called A.f()
Called B.f()
```

Частичное переопределение атрибутов

```
>>> class A(object):
>>>     NAME = "A"
>>>
>>>     def f(self):
>>>         print(self.NAME)
>>>
>>> class B(A):
>>>     NAME = "B"
>>>
>>> a, b = A(), B()
>>> a.f()
>>> b.f()
```

A

B

Переопределение конструктора

```
>>> class A(object):
>>>     def __init__(self):
>>>         self.x = 1
>>>
>>> class B(A):
>>>     def __init__(self):
>>>         self.y = 2
>>>
>>> b = B()
>>> print(b.x)
AttributeError: ...

>>> print(b.y)
2
```

ВЫЗОВ МЕТОДОВ БАЗОВОГО КЛАССА

Конструктор базового класса также должен был быть вызван. Для этого используют одно из двух обращений:

- `super(class_, self).method(...)`
- `class_.method(self, ...)`

Второй вариант нежелателен, однако порой необходим при *множественном* наследовании.

Замечание. В Python 3 метод **super** внутри класса можно вызывать без параметров – будут использованы значения по умолчанию.

Замечание. Метод **super** возвращает специальный проху-объект, а потому обратиться к некоторым «магическим» методам. Например, обратиться по индексу к нему невозможно.

ВЫЗОВ КОНСТРУКТОРА БАЗОВОГО КЛАССА

```
>>> class A(object):
>>>     def __init__(self):
>>>         self.x = 1
>>>
>>> class B(A):
>>>     def __init__(self):
>>>         super(B, self).__init__()
>>>         # A.__init__(self) - второй нежелательный вариант
>>>         self.y = 2
>>>
>>> b = B()
>>> print b.x, b.y
1 2
```

Магические методы

Методы со специальными именами вида `__*__` называют магическими. Они отвечают за многие операции с объектом. Список магических методов можно увидеть в описании `DataModel` ([Py2](#), [Py3](#)), а также на странице модуля **operator**.

Создание, инициализация, удаление класса: *new, init, del*

Приведение типа:

- к строке – *repr, str/unicode* (Py2) или *bytes/str* (Py3)
- к **bool** – *nonzero* (Py2) или *bool* (Py3)

Замечание. Преобразовать, к строке объект можно вызвав **str(x)** или `x.__str__()`, к **bool** – **bool(x)** или `x.__bool__()` (*nonzero*).

Замечание. Если метод преобразования к **bool** не реализован, возвращается результат метода `__len__`, а если и его нет, то все объекты преобразуются к **True**.

Сравнение и хеширование

Сравнение и хеширование: *eq*, *ne*, *le*, *lt*, *ge*, *gt* и *hash*.

Замечание. Если метод *eq* для двух объектов возвращает **True**, то *hash* объектов должен также совпадать.

Замечание. Если класс не реализует некоторое сравнение, он может вернуть **NotImplemented**.

Замечание. Явно реализовывать все операторы не нужно. Достаточно метода *eq* и одного из методов сравнения, а также декоратора **functools.total_ordering**.

Операции с числами

Можно переопределить любые математические операции: + (**add**), - (**sub**), * (**mul**), @ (**matmul**), / (**truediv**), // (**floordiv**), и прочие.

Помимо таких операций есть еще методы-компаньоны. Например, для сложения они называются **radd** и **iadd**. Метод **radd** вызывается, когда у левого операнда метод **add** не реализован, а **iadd** – для операции “+=”.

Также есть возможность переопределить операции приведения к типам **complex**, **float** и **int**, округления **round** и взятия модуля **abs**.

Прочие методы

Метод **call** переопределяет оператор вызова `()` (круглые скобки).

Метод **len** – взятие длины (может вызываться как `len(.)`; в Python 3 есть также `length_hint`).

Методы **getitem**, **setitem**, **delitem** – работа с индексами (оператор `[]`).

Методы **iter**, **reversed**, **contains** отвечают за итерирование и проверку вхождения.

Методы **instancecheck** и **subclasscheck** отвечают за проверку типа.

Метод **missing** вызывается словарем, если запрошенный ключ отсутствует (переопределен в `defaultdict`).

Декораторы

...

Декораторы

Декораторы ([PEP-318](#)):

- Выполняют некоторое дополнительное действие при вызове или создании функции
- Модифицируют функцию после создания
- Могут принимать аргументы
- Упрощают написание кода

Рассмотрим пример декоратора – функции, которая при вызове декорированной функции проверяет, возвращенное ей значение имеет тип *float*. Функцию-декоратор назовем *check_return_type_float*.

Пример реализации

Пример использования (проверяет, что возвращаемое значение типа *float*):

```
>>> @check_return_type_float
>>> def g():
>>>     return 'not a float'
```

Эквивалентной записью будет следующая:

```
>>> def g():
>>>     return 'not a float'
>>>
>>> g = check_return_type_float(g)
```

Пример реализации

Декоратор реализован как функция, которая возвращает другую функцию – *wrapper*:

```
>>> def check_return_type_float(f):  
>>>     def wrapper(*args, **kwargs):  
>>>         result = f(*args, **kwargs)  
>>>         assert isinstance(result, float)  
>>>         return result  
>>>     return wrapper
```

Нюансы декорирования

Поскольку декоратор возвращает другую функцию, в примере `check_return_type_float` у переменной `f` будет имя `'wrapper'`.

Для того, чтобы метаданные (имя, документация) были корректными, внутренней функции (`wrapper`'у) добавляют декоратор **`functools.wraps`**:

```
>>> def check_return_type_float(f):  
>>>     @functools.wraps(f)  
>>>     def wrapper(*args, **kwargs):  
>>>         # some code  
>>>     return wrapper
```

Замечание. В Python 3 декорировать можно не только функции, но и классы ([PEP-3129](#)).

Реализация декоратора с ПОМОЩЬЮ КЛАССА

```
>>> class FloatTypeChecker(object):
>>>     def __init__(self, f):
>>>         self.f = f
>>>
>>>     def __call__(self, *args, **kwargs):
>>>         result = self.f(*args, **kwargs)
>>>         assert isinstance(result, float)
>>>         return result
>>>
>>> check_return_type_float = FloatTypeChecker
```

Вопрос: Как применить здесь **functools.wraps**?

Замечание. Добавлять данный декоратор желательно везде. Это и хороший стиль кода, и так остается возможность узнать имя вызванной функции run-time.

Декораторы с параметрами

Для создания более общего декоратора, логично ему добавить возможность принимать параметры.

```
>>> def check_return_type(type_):
>>>     def wrapper(f):
>>>         @functools.wraps(f)
>>>         def wrapped(*args, **kwargs):
>>>             result = f(*args, **kwargs)
>>>             assert isinstance(result, type_)
>>>             return result
>>>         return wrapped
>>>     return wrapper
>>>
>>> @check_return_type(float)
>>> def g():
>>>     return 'not a float'
```

Несколько декораторов

Эквивалентной записью будет следующая:

```
>>> def g():  
>>>     return 'not a float'  
>>>  
>>> g = check_return_type(float)(g)
```

Допустимо применять несколько декораторов – один над другим.

```
>>> @decorator2  
>>> @decorator1  
>>> def f():  
>>>     pass
```

Эквивалентная запись применения декораторов к функции **f** имеет вид:

```
>>> f = decorator2(decorator1(f))
```

Параметризованный декоратор-класс

Декораторы с параметрами можно реализовать как класс. В таком случае параметры будут сохраняться в методе `__init__`, а декорированную функцию следует возвращать в `__call__`.

```
>>> class FloatTypeChecker(object):
>>>     def __init__(self, result_type):
>>>         self._result_type = result_type
>>>
>>>     def __call__(self, f):
>>>         @functools.wraps(f)
>>>         def wrapper(*args, **kwargs):
>>>             result = f(*args, **kwargs)
>>>             assert isinstance(result, self._result_type)
>>>             return result
>>>     return wrapper
```


Отвѣты на вопросы

...

ОТВЕТЫ НА ВОПРОСЫ

Очистить список, не удалив его самого, можно так:

```
>>> x = []  
>>> del x[:]  
>>> x[:] = []    # эквивалентная запись
```

Пустая lambda-функция имеет следующий вид:

```
lambda: None
```

При создании декораторов-классов для применения **functools.wraps** обычно переопределяют метод `__new__`. Применить напрямую к классу его не удастся. Второй вариант – применить **functools.update_wrapper** в методе `__init__` ко вновь созданному объекту класса.